

Hypergraph Acyclicity and Propositional Model Counting

Florent Capelli* Arnaud Durand† Stefan Mengel‡

January 27, 2014

We show that the propositional model counting problem $\#SAT$ for CNF-formulas with hypergraphs that allow a disjoint branches decomposition can be solved in polynomial time. We show that this class of hypergraphs is incomparable to hypergraphs of bounded incidence cliquewidth which were the biggest class of hypergraphs for which $\#SAT$ was known to be solvable in polynomial time so far. Furthermore, we present a polynomial time algorithm that computes a disjoint branches decomposition of a given hypergraph if it exists and rejects otherwise. Finally, we show that some slight extensions of the class of hypergraphs with disjoint branches decompositions lead to intractable $\#SAT$, leaving open how to generalize the counting result of this paper.

1 Introduction

Proposition model counting ($\#SAT$) is the problem of counting satisfying assignments (models) to a CNF-formula. It is the canonical $\#P$ -hard counting problem and is important due to its applications in Artificial Intelligence. Unfortunately, $\#SAT$ is extremely hard to solve: Even on restricted classes of formulas like monotone 2CNF-formulas or Horn 2CNF-formulas it is NP -hard to approximate within a factor of $2^{n^{1-\epsilon}}$ for any $\epsilon > 0$ [Rot96]. Fortunately, this is not the end of the story: While syntactical restrictions on the types of allowed clauses do not lead to tractable counting, there is a growing body of work that successfully applies so-called *structural* restrictions to $\#SAT$, see e.g. [FMR08, SS10, PSS13, SS14]. In this line of work one does not restrict the

*IMJ UMR 7586 - Logique, Université Paris Diderot, France, Email: fcapelli@math.univ-paris-diderot.fr

†IMJ UMR 7586 - Logique, Université Paris Diderot and LSV UMR 8643, ENS Cachan, France, Email: durand@math.univ-paris-diderot.fr

‡Laboratoire d'Informatique, LIX UMR 7161, Ecole Polytechnique, France, Email: mengel@lix.polytechnique.fr. Partially supported by DFG grants BU 1371/2-2 and BU 1371/3-1.

individual clauses of CNF-formulas but instead the interaction between the variables in the formula. This is done by assigning graphs or hypergraphs to formulas and then restricting the class of (hyper)graphs that are allowed for instances (see Section 2 for details). In this paper we present a new class of hypergraphs, such with disjoint branches decompositions [Dur12], for which $\#SAT$ is tractable.

Having a disjoint branches decomposition is a so-called acyclicity notion for hypergraphs. Unlike for graphs, there are several reasonable ways of defining acyclicity for hypergraphs [Fag83] which have been very successful in database theory. Mostly three “degrees of acyclicity” have been studied: α -acyclicity, β -acyclicity and γ -acyclicity, where the α -acyclic hypergraphs form the most general and the γ -acyclic hypergraphs the least general class. Prior to this paper it was known that $\#SAT$ for CNF-formulas with α -acyclic hypergraphs was $\#P$ -hard [SS10], while it is tractable for γ -acyclic hypergraphs as the latter have incidence cliquewidth bounded by 3 [GP04] and thus the results of [SS14] apply.

To understand the influence of hypergraph acyclicity on the complexity of $\#SAT$, the next natural step is thus analyzing the intermediate case of β -acyclic hypergraphs. For this class it is known that SAT is tractable [OPS13], unlike for α -acyclic hypergraphs. Unfortunately, the algorithm in [OPS13] is based on a resolution-like method and it is not clear whether one can obtain tractability for counting from the method used for decision. In fact, most classical decision results based on tractability of resolution (such as for 2-SAT) or unit propagation (Horn-SAT) do not extend to counting as the respective counting problems are hard (see e.g. [Rot96]).

Unfortunately, $\#SAT$ for CNF-formulas with β -acyclic hypergraphs has turned out to be a stubborn problem whose complexity could so far not be determined despite considerable effort by us and others [Sli14]. A natural approach which we follow in this paper is thus trying to understand slightly more restrictive notions of acyclicity. We focus here on hypergraphs with disjoint branches decompositions, a notion which was introduced by Duris [Dur12] and which lies strictly between β -acyclicity and γ -acyclicity. We show that for CNF-formulas whose hypergraphs have a disjoint branches decompositions we can solve $\#SAT$ in polynomial time. We also show that hypergraphs with disjoint branches decompositions are incomparable to hypergraphs with bounded incidence cliquewidth which so far were the biggest class of hypergraphs for which $\#SAT$ was known to be tractable. Thus our results give a new class of tractable instances for $\#SAT$, pushing back the known tractability frontier for this problem.

Our main contribution is twofold: Most importantly, we present the promised counting algorithm for CNF-formulas whose hypergraphs have a disjoint branches decomposition in Section 3. Secondly, we present in Section 4 a polynomial time algorithm that checks if a hypergraph has a disjoint branches decomposition and if so also constructs it. On the one hand, this gives some confidence that hypergraphs with disjoint branches decompositions form a well-behaved class as it can be decided in polynomial time. On the other hand, the counting algorithm will depend on knowing a decomposition, so its computation is an essential part of the counting procedure. Finally, in Section 5 we then turn to generalizing the results of this paper, unfortunately showing only negative results. We consider some natural looking extensions of hypergraphs with disjoint branches and

show that $\#SAT$ is intractable on these classes under standard complexity theoretic assumptions.

2 Preliminaries and notation

2.1 Hypergraphs and graphs associated to CNF-formulas

In this section we describe graphs and hypergraphs commonly associated to CNF-formulas and introduce restricted classes of hypergraphs that we will consider in this paper. The *primal graph* of a CNF-formula F has as vertices the variables of F and two vertices are connected by an edge if they appear in a common clause of F . The *incidence graph* of F is defined as the bipartite graph which has as vertices the variables and the clauses of F and two vertices u and v are connected by an edge if u is a variable and v is a clause such that u appears in v . The *signed incidence graph* of such a formula is obtained from its incidence graph by orientating edges to indicate positive or negative occurrences of variables in clauses (see [FMR08] for details).

A (finite) hypergraph \mathcal{H} is a pair (V, E) where V is a finite set and $E \subseteq \mathcal{P}(V)$. A subhypergraph $\mathcal{H}' = (V', E')$ of $\mathcal{H} = (V, E)$ is a hypergraph with $V' \subseteq V$ and $E' \subseteq \{e \cap V' \mid e \in E, e \cap V' \neq \emptyset\}$. A path between two vertices $u, v \in V$ is defined to be a sequence e_1, \dots, e_k such that $u \in e_1$, $v \in e_k$ and for every $i = 1, \dots, k-1$ we have $e_i \cap e_{i+1} \neq \emptyset$. A hypergraph \mathcal{H} is called *connected* if there is a path between every pair of vertices of \mathcal{H} . A (connected) *component* of \mathcal{H} is defined to be a maximal connected subhypergraph of \mathcal{H} .

To a CNF-formula F we associate a hypergraph $\mathcal{H} = (V, E)$ where V is the variable set of F and the hyperedge set E contains for each clause of F an edge containing the variables of the clause.

2.1.1 Graph Decompositions

We will not recall basic graph decompositions such as tree-width and clique-width (see e.g. [GP04, FMR08]). A class of CNF-formulas is defined to be of bounded (signed) incidence clique-width, if their (signed) incidence graphs are of bounded clique-width. A set $X \subseteq V$ of vertices of a graph is called a *module*, if every $v \in V \setminus X$ has the same set of neighbours and non-neighbours in X . Intuitively, the elements of a module X are indiscernible by vertices outside of X . If X is a module of a graph G , the graph $G' = (V', E')$ obtained after contraction of X is defined by $V' := (V \setminus X) \cup \{x\}$ where x is a new vertex not in V and $E' := (E \cap (V \setminus X)^2) \cup \{ux : u \notin X \text{ and } \exists v \in X \text{ s.t. } uv \in E\}$. A class of CNF-formulas is of bounded *modular incidence treewidth* if their incidence graphs are of bounded tree-width after contracting all modules.

2.1.2 Acyclicity in hypergraphs

It is well-known that, in contrast to the graph setting, there are several non equivalent notions of acyclicity for hypergraphs [Fag83]. Most of these notions have many equivalent

definitions (see [Fag83, Dur12] for elimination rule based or cycle based definitions, for example), but we will mainly restrict ourselves to definitions using the notion of join trees.

Definition 1. A join tree of a hypergraph $\mathcal{H} = (V, E)$ is a pair (\mathcal{T}, λ) where $\mathcal{T} = (N, T)$ is a tree and λ is a bijection between N and E such that:

- for each $e \in E$, there is a $t \in N$ such that $\lambda(t) = e$, and
- for each $v \in V$, the set $\{t \in N \mid v \in \lambda(t)\}$ is a connected subtree of \mathcal{T} .

The second condition in Definition 1 is often called the *connectedness condition*. It is often convenient to identify an edge $e \in E$ with the vertex $\lambda(e) \in N$ and we will mostly follow this convention in this paper. We call a join tree (\mathcal{T}, λ) a *join path* if the underlying tree \mathcal{T} is a path.

A hypergraph is defined to be α -acyclic if it has a join tree [Fag83]. This is the most general acyclicity notion for hypergraphs commonly considered. However, α -acyclicity is not closed under taking subhypergraphs: an α -acyclic hypergraph may have cyclic subhypergraphs. To remedy this situation, one considers the restricted notion of β -acyclicity where a hypergraph is defined to be β -acyclic if it is α -acyclic and all of its subhypergraphs are also α -acyclic.

A γ -cycle in a hypergraph is a sequence $(e_1, x_1, \dots, e_n, x_n)$ with $n \geq 3$ where the x_i are distinct vertices and the e_i are distinct hyperedges such that,

- for all $i \in [1, \dots, n-1]$, x_i belongs to e_i and e_{i+1} and to no other e_j for $j \neq i, i+1$.
- x_n belongs to e_n and e_1 and to possibly to other e_j s.

A hypergraph is γ -acyclic if it has no γ -cycle. This notion can also be characterized and generalized through the notion of disjoint branches decompositions.

Definition 2. A disjoint branches decomposition of a hypergraph \mathcal{H} is a join tree (\mathcal{T}, λ) such that for every two nodes t and t' appearing on different branches of \mathcal{T} we have $\lambda(t) \cap \lambda(t') = \emptyset$.

Disjoint branches decompositions were introduced by Duris [Dur12] who proved that a hypergraph is γ -acyclic if and only if it has a disjoint branches decomposition for any choice of hyperedge as a root. Furthermore, he showed that every hypergraph with a disjoint branches decomposition is β -acyclic.

2.2 Known complexity results and comparisons between classes

We show the known complexity results for the restrictions of #SAT we have introduced before in Table 2.2; for definitions of the appearing complexity classes see e.g. [FG06].

The four acyclicity notions and classes defined by bounding the introduced width measures form a hierarchy for inclusion which is depicted in Figure 1. Most of the proofs of inclusion can be found in [Fag83, Dur12, GP04, PSS13] and the references therein. We give the missing results in this sections.

| class | lower bound | upper bound |
|------------------------------|---------------------------|------------------------|
| primal treewidth | | FPT [SS10] |
| incidence treewidth | | FPT [SS10] |
| modular incidence treewidth | | FPT [PSS13] |
| signed incidence cliquewidth | | FPT [FMR08] |
| incidence cliquewidth | W[1] -hard [OPS13] | XP [SS14] |
| γ -acyclic | | FP [GP04, SS14] |
| β -acyclic | ? | ? |
| α -acyclic | #P -hard [SS10] | #P |
| disjoint branches | | FP (this paper) |

Table 1: Known complexity results for structural restrictions of #SAT.

γ -acyclicity and modular treewidth are incomparable. We exhibit a family of γ -acyclic hypergraph whose associated incidence graph have unbounded modular treewidth. Let $n \in \mathbb{N}$ and \mathcal{H}_n be the hypergraph of vertex set $\{x_1, \dots, x_n, y_1, \dots, y_n\}$ and of hyperedge set

$$\{\{y_i, x_1, \dots, x_n\} \mid i \leq n\} \cup \{\{x_i\} \mid i \leq n\}.$$

Clearly, \mathcal{H}_n is γ -acyclic. Also, the incidence graph of \mathcal{H}_n has no modules because of the y_i and the singleton edges $\{x_i\}$. Thus the treewidth of \mathcal{H}_n and its modular treewidth coincide. Furthermore, it is easy to see that the incidence graph of \mathcal{H}_n contains a subgraph that is isomorphic to $K_{n,n}$. Since treewidth is stable under taking subgraphs and $K_{n,n}$ is well-known to have treewidth n , it follows that the hypergraphs \mathcal{H}_n have unbounded modular treewidth.

For the other direction, cycles have bounded modular treewidth but are not γ -acyclic.

Disjoint branches and incidence clique-width are incomparable. In this section we will show that unlike γ -acyclic hypergraphs the hypergraphs with disjoint branches decompositions have unbounded cliquewidth. In fact we will even show this for hypergraphs with join paths. Since join paths do not branch, these hypergraphs are a subclass of the hypergraphs with disjoint branches decompositions.

We will use the following characterization of hypergraphs with join paths.

Lemma 3. *A hypergraph $\mathcal{H} = (V, E)$ has a join path if and only if there exists an order $<_E$ on the edge set E of \mathcal{H} such that for all $e, f, g \in E$ such that $e <_E f <_E g$, if $v \in e \cap g$ then $v \in f$.*

Proof. If \mathcal{P} is a join path of \mathcal{H} , we choose an orientation of this path and then define $e <_E f$ if and only if e appears before f in \mathcal{P} . If $e <_E f <_E g$ and $v \in e \cap g$, then as f is between e and g in \mathcal{P} . From the connectedness condition of v , we get $v \in f$.

For the other direction, let $<_E$ be an order on E . Let $E := \{e_1, \dots, e_n\}$ with $e_i <_E e_{i+1}$ for $i < n - 1$. Let \mathcal{P} be the path whose vertices are E and edges are (e_i, e_{i+1}) for $i < n$. We claim that \mathcal{P} is a join path of \mathcal{H} . Obviously \mathcal{P} is a path, so we only have to show

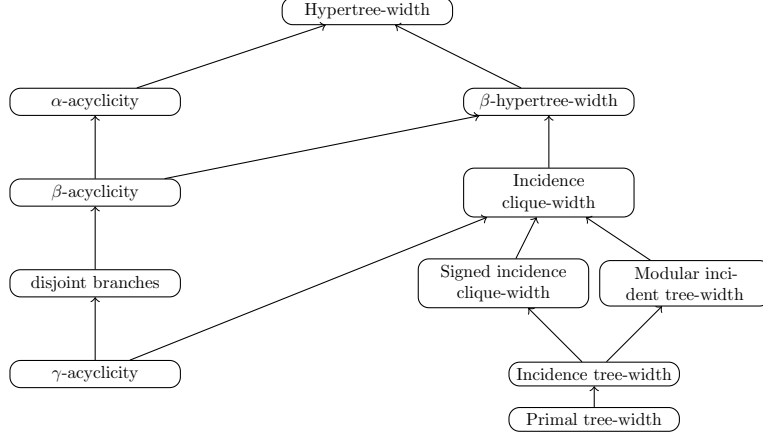


Figure 1: A hierarchy of inclusion of graph and hypergraph classes. Classes not connected by a directed path are incomparable.

the connectedness property. Let $v \in V$, then for all $i \leq k \leq j$, if $v \in e_i \cap e_j$, then $v \in e_k$ by assumption on $<_E$. So the edges containing v are connected in \mathcal{P} which proves the claim. \square

Definition 4. Let $G = (X, Y, E)$ be a bipartite graph. A strong ordering $(<_X, <_Y)$ of G is a pair of orderings on X and Y such that for all $x, x' \in X$ and $y, y' \in Y$, such that $x <_X x'$ and $y <_Y y'$, if $(x, y) \in E$ and $(x', y') \in E$, then $(x, y') \in E$ and $(x', y) \in E$. G is called a bipartite permutation graph if it admits a strong ordering.

Brandstädt and Lozin showed the following property of bipartite permutation graphs.

Lemma 5 ([BL03]). *Bipartite permutation graphs have unbounded cliquewidth.*

It turns out that hypergraphs with a bipartite permutation incidence graph admit a join path.

Proposition 6. *Every hypergraph \mathcal{H} without empty hyperedges, whose incidence graph \mathcal{H}_I is a bipartite permutation graph, has a join path.*

Proof. Let $(<_V, <_E)$ be a strong ordering of $\mathcal{H}_I = (V, E, A)$. We claim that for all $e <_E f <_E g$, if $v \in e \cap g$ then $v \in f$. Indeed, as f is not empty, there exists $w \in f$. If $w = v$, there is nothing to prove. Otherwise, suppose that $w <_V v$. Then, by definition of strong orderings, as $(f, w) \in A$ and $(g, v) \in A$, we have $(f, v) \in A$. Thus $v \in f$. The case $v <_V w$ follows symmetrically: $(f, w) \in A$ and $(e, v) \in A$ implies that $(f, v) \in A$ so $v \in f$. Thus the ordering $<_E$ has the property of lemma 3 and it follows that \mathcal{H} has a join path. \square

By combining Lemma 5 and Proposition 6 we get:

Corollary 7. *The class of CNF-formulas with join paths has unbounded incidence cliquewidth. The same is thus true for CNF-formulas with disjoint branches decompositions.*

Again, cycles give an example proving the other direction of incomparability.

2.3 Representation of #SAT by constraint satisfaction problems

It will be convenient to describe our counting algorithm in the framework of constraint satisfaction problems in negative representation [CGH09]. We will discuss below how this representation relates to #SAT.

Let D be a finite set called domain. A constraint $C = (R, \bar{x})$ is a pair where $R \subseteq D^r$ is a relation and $\bar{x} = (x_{i_1}, \dots, x_{i_r})$ is a list of variables of length r . An instance Φ of the (uniform) constraint satisfaction problem, is a set of constraints. We denote by $\text{var}(\Phi)$ the set X of variables of Φ . The instance Φ is satisfied by an assignment $a : \text{var}(\Phi) \rightarrow D$ if for all $(R, \bar{x}) \in \Phi$ we have $a(\bar{x}) = (a(x_{i_1}), \dots, a(x_{i_r}))$ is in the relation R . We denote this by $a \models \Phi$.

The associated counting problem, #CSP, is, given an instance Φ , to compute

$$|\{a \mid a \models \Phi\}|,$$

i.e., the number of satisfying assignments of Φ .

No hypothesis is made above on the arity of relations which is not a priori bounded and may differ for different relations. So, it may be more succinct to represent each relation R by listing the tuples in its complement $R^c := D^r \setminus R$. Consequently, we define the counting constraint satisfaction problem in negative representation, #CSP_{neg}, that is to compute, given Φ where each relation R is encoded by listing the tuples in R^c , the number of satisfying assignments of Φ .

The relation to #SAT: It is natural to represent CNF-formulas by a Boolean CSP_{neg}-instance. Indeed, as we do not bound the length k of clauses, it is more realistic to represent each associated constraint relation not by its set of $2^k - 1$ models (as common in the area of constraint satisfaction) but by its complement containing the unique counter model of the clause.

In the other direction one can easily encode every #CSP_{neg}-instance by a CNF-formula (see also [BB12]): In a first step encode all domain elements in binary, introducing vertex modules in the hypergraph. Then we encode every tuple in every relation by a clause that disallows the respective tuple. Observe that after the contraction of some modules, the CNF-formula has the same hypergraph as the original #CSP_{neg}-instance. Since the class of hypergraphs with disjoint branches decompositions is stable under introducing or contracting of modules, it follows that #SAT and #CSP_{neg} are equivalent for our considerations.

3 Counting solutions of disjoint branches queries

In this section we will show that $\#CSP_{\text{neg}}$ —and thus also $\#SAT$ —restricted to hypergraphs with a disjoint branches decomposition can be solved in polynomial time. It will be convenient to work with inputs of the form $\phi = R_1 \vee \dots \vee R_k$, i.e., we will count assignments a that satisfy at least one of the R_i . By basic Boolean algebra and inclusion-exclusion, solving instances of this type is equivalent to $\#CSP_{\text{neg}}$. Observe that when transforming a $\#CSP_{\text{neg}}$ -instance into the disjunctive form, this changes the encoding of the relations from negative representation to positive representation, i.e., we list the tuples in the relations R_i explicitly and *not* those in the complement.

Note that, as discussed in Section 2.3, we may assume that the domain of all relations is $\{0, 1\}$, so we restrict ourselves to this case.

Let us introduce some notation: Let X, Y be two sets of variables and let $a: X \rightarrow \{0, 1\}$ and $b: Y \rightarrow \{0, 1\}$ be two assignments. We call a and b *consistent*, symbol $a \sim b$, if they agree on their common variables $X \cap Y$. If, in addition, $X \subseteq Y$, we write $a \subseteq b$. Finally, if a and b have disjoint domains, i.e., $X \cap Y = \emptyset$, we denote by $a \oplus b$ the assignment on $X \cup Y$ defined in the obvious way.

For an assignment $a: X \rightarrow \{0, 1\}$ and a set of variables $Y \subseteq X$ the restriction of a onto Y is denoted by $a|_Y$.

Let ϕ be an instance and let ψ be a subformula of ϕ . Let X a subset of the variables such that $\text{var}(\psi) \subseteq X \subseteq \text{var}(\phi)$ and let a be a partial assignment of variables in $\text{var}(\phi)$. We denote by $Sol_X(\psi, a) = \{b : X \rightarrow \{0, 1\}, b \models \psi, a \sim b\}$ and by $S_X(\psi, a) = |Sol_X(\psi, a)|$. The number of solutions of ϕ is then $S_{\text{var}(\phi)}(\phi, \emptyset)$, where \emptyset denotes the empty partial assignment. We show that by computing a polynomial number of values $S_X(\psi, a)$ in polynomial time, we can compute $S_{\text{var}(\phi)}(\phi, \emptyset)$. To this end we will prove several lemmas that will allow us a recursive computation.

The first lemma shows how the disjointness naturally appears when we want to count solutions:

Lemma 8. *Let ϕ_1 and ϕ_2 be two instances and $X \supseteq \text{var}(\phi_1) \cup \text{var}(\phi_2)$, then*

$$S_X(\phi_1 \vee \phi_2, a) = S_X(\phi_1, a) + S_X(\phi_2, a) - |Sol_X(\phi_1, a) \cap Sol_X(\phi_2, a)|.$$

Proof. Using inclusion-exclusion and the fact that $Sol_X(\phi_1 \vee \phi_2, a) = Sol_X(\phi_1, a) \cup Sol_X(\phi_2, a)$ directly yields the result. \square

The next lemma will allow us to efficiently compute $Sol_X(\phi_1 \vee \phi_2, a)$ recursively.

Lemma 9. *Let $X_1 = \text{var}(\phi_1)$ and $X_2 = \text{var}(\phi_2)$. Assume that $X_1 \cap X_2 = \emptyset$ and let $X = X_1 \cup X_2$. Let a be a partial assignment of the variables of X and $a_1 = a|_{X_1}$, $a_2 = a|_{X_2}$. Then $Sol_X(\phi_1, a) \cap Sol_X(\phi_2, a) = \{b_1 \oplus b_2 \mid b_i : X_i \rightarrow \{0, 1\}, b_i \models \phi_i, a_i \subseteq b_i\}$ and $|Sol_X(\phi_1, a) \cap Sol_X(\phi_2, a)| = S_{X_1}(\phi_1, a_1)S_{X_2}(\phi_2, a_2)$.*

Proof. The inclusion from right to left is trivial. For the other inclusion, it is enough to remark that for $b \in Sol_X(\phi_1, a) \cap Sol_X(\phi_2, a)$, we have $b = b|_{X_1} \oplus b|_{X_2}$ and $a_i \subseteq b|_{X_i}$ as $a \subseteq b$. The equality of the sizes follows directly. \square

We now show how we can add variables that do not appear in ϕ .

Lemma 10. *Let $Y \subseteq X$ and $a : X_0 \rightarrow \{0, 1\}$ for $X_0 \subseteq X$. Then*

$$S_X(\phi, a) = 2^{|X \setminus (Y \cup X_0)|} S_Y(\phi, a|_Y).$$

Proof. Every solution of ϕ on Y can be arbitrarily extended on $X \setminus (X_0 \cup Y)$ and thus the claim follows directly. \square

The next corollary lets us handle the disjunction of more than two terms.

Corollary 11. *Let ϕ_1, \dots, ϕ_k be formulas with $X_i = \text{var}(\phi_i)$ and $X_i \cap X_j = \emptyset$ for every combination $i, j \in [k], i \neq j$. Let X be a set such that $X_1 \cup \dots \cup X_k \subseteq X$ and let $a : X_0 \rightarrow \{0, 1\}$ for $X_0 \subseteq X$, $a_i = a|_{X_i}$. Then*

$$S_X\left(\bigvee_{j=1}^k \phi_j, a\right) = 2^{|X \setminus (X_0 \cup X_1 \cup \dots \cup X_k)|} \sum_{i=1}^k S_{X_i}(\phi_i, a_i) \prod_{j=1}^{i-1} (2^{|X_j \setminus X_0|} - S_{X_j}(\phi_j, a_j)) \prod_{j=i+1}^k 2^{|X_j \setminus X_0|}$$

Proof. The proof proceeds by induction on k . For $k = 1$, we have to show $S_X(\phi_1, a) = 2^{|X \setminus (X_0 \cup X_1)|} S_{X_1}(\phi_1, a_1)$ which is Lemma 10.

Now let $Z = X_1 \cup \dots \cup X_{k+1}$ and $Z_1 = Z \setminus X_1$. To ease notations, we denote by a' the assignment $a|_Z$. Remark that $a_i = a|_{X_i} = a'|_{X_i}$ and $a|_{Z_1} = a'|_{Z_1}$ as $X_i \subseteq Z$.

As X_1 is disjoint from Z_1 , we can apply Lemma 9 to ϕ_1 and $\phi_2 \vee \dots \vee \phi_{k+1}$ to get

$$S_Z\left(\bigvee_{j=1}^{k+1} \phi_j, a'\right) = S_Z(\phi_1, a') + S_Z\left(\bigvee_{j=2}^{k+1} \phi_j, a'\right) - S_{X_1}(\phi_1, a_1) S_{Z_1}\left(\bigvee_{j=2}^{k+1} \phi_j, a|_{Z_1}\right).$$

By applying Lemma 10 and distributivity, we get

$$S_Z\left(\bigvee_{j=1}^{k+1} \phi_j, a'\right) = 2^{|Z_1 \setminus X_0|} S_{X_1}(\phi_1, a_1) + S_{Z_1}\left(\bigvee_{j=2}^{k+1} \phi_j, a|_{Z_1}\right) (2^{|X_1 \setminus X_0|} - S_{X_1}(\phi_1, a_1)).$$

Now we use the induction hypothesis on $\bigvee_{j=2}^{k+1} \phi_j$ and the equality $2^{|Z_1 \setminus X_0|} = \prod_{j=2}^{k+1} 2^{|X_j \setminus X_0|}$ to get

$$\begin{aligned} & S_Z\left(\bigvee_{j=1}^{k+1} \phi_j, a'\right) \\ &= \prod_{j=2}^{k+1} 2^{|X_j \setminus X_0|} S_{X_1}(\phi_1, a_1) \\ & \quad + (2^{|X_1 \setminus X_0|} - S_{X_1}(\phi_1, a_1)) \left(\sum_{i=2}^{k+1} S_{X_i}(\phi_i, a_i) \prod_{j=2}^{i-1} (2^{|X_j \setminus X_0|} - S_{X_j}(\phi_j, a_j)) \prod_{j=i+1}^{k+1} 2^{|X_j \setminus X_0|} \right) \\ &= \sum_{i=1}^{k+1} S_{X_i}(\phi_i, a_i) \prod_{j=1}^{i-1} (2^{|X_j \setminus X_0|} - S_{X_j}(\phi_j, a_j)) \prod_{j=i+1}^{k+1} 2^{|X_j \setminus X_0|}. \end{aligned}$$

Finally, applying Lemma 10 yields

$$S_X\left(\bigvee_{j=1}^{k+1} \phi_j, a\right) = 2^{|X \setminus (X_0 \cup X_1 \cup \dots \cup X_{k+1})|} \sum_{i=1}^{k+1} S_{X_i}(\phi_i, a_i) \prod_{j=1}^{i-1} (2^{|X_j \setminus X_0|} - S_{X_j}(\phi_j, a_j)) \prod_{j=i+1}^{k+1} 2^{|X_j \setminus X_0|}.$$

which is the desired result. \square

A final lemma will help us to compute the size of the intersections of the solutions to a formula and a single relation.

Lemma 12. $|Sol_X(R, a) \cap Sol_X(\phi, a)| = \sum_{b \in Sol_X(R, a)} S_X(\phi, b)$

Proof. This follows from the fact that $Sol_X(R, a) \cap Sol_X(\phi, a) = \bigcup_{b \in Sol_X(R, a)} Sol_X(\phi, b)$ and that the union is disjoint. \square

We now finally show the main result of this section.

Theorem 13. *There is a polynomial time algorithm that, given an instance $\phi = \bigvee_{i=1}^m R_i$ and a disjoint branches decomposition of the hypergraph of ϕ , computes the number of satisfying assignments of ϕ .*

Proof. Let $\phi = R_1 \vee \dots \vee R_m$ be an instance with hypergraph \mathcal{H} . Let $r_i := |R_i|$ and let $r := \sum_{i=1}^m r_i$. Let furthermore (\mathcal{T}, λ) be a disjoint branches decomposition of \mathcal{H} . For a vertex t of \mathcal{T} , we denote by \mathcal{T}_t the subtree of \mathcal{T} rooted in t and by ϕ_t the associated subinstance. Furthermore, R_t is defined to be the relation associated to t . Finally, we denote by V_t the set of variables of ϕ_t .

We will give a polynomial time algorithm that computes inductively from the leaves to the root of \mathcal{T} certain values $S_X(\psi, a)$ where ψ is a subinstance of ϕ . Our goal is to compute $S_{V_r}(\phi_r, \emptyset)$ where r is the root of \mathcal{T} , since this value is the number of solutions of ϕ . More precisely, for a given t in the tree, we compute $S_{V_t}(\phi_t, \emptyset)$ and for all ancestors u of t and all $b \in R_u$, we compute $S_{V_t}(\phi_t, b|_{\text{var}(R_t)})$. Since there are m vertices in \mathcal{T} and at most $r + 1$ values to compute for each vertex, we will to compute at most $m(r + 1)$ different values. We will show how to compute these values in polynomial time to get a polynomial time algorithm overall.

Let t be a vertex of \mathcal{T} , u one of his ancestors and $b \in R_u$. If t is a leaf, then ϕ_t consists only of the relation R_t . Then $Sol_{V_t}(\phi_t, \emptyset) = R_t$ and $Sol_{V_t}(\phi_t, b|_{\text{var}(R_t)}) = \{a \in R_t \mid a \sim b\}$ so the computations can be done efficiently.

Now that assume t has children t_1, \dots, t_k . To ease notation, let $V_i = V_{t_i}$, $\phi_i = \phi_{t_i}$ and $R_i = R_{t_i}$. Observe that by Lemma 8 we have

$$S_{V_t}(\phi_t, \emptyset) = S_{V_t}(R_t, \emptyset) + S_{V_t}\left(\bigvee_{j=1}^k \phi_j, \emptyset\right) - |Sol_{V_t}(R_t, \emptyset) \cap Sol_{V_t}\left(\bigvee_{j=1}^k \phi_j, \emptyset\right)|.$$

As the variables of the ϕ_i are disjoint, by Corollary 11, one can compute $S_{V_t}(\bigvee_{j=1}^k \phi_j, \emptyset)$ in $O(k)$ if the values of $Sol_{V_i}(\phi_i, \emptyset)$ are precomputed, which is the case by induction.

In addition, $S_{V_t}(R_t, \emptyset) = 2^{|V_t \setminus \text{var}(R_t)|} |R_t|$ since a solution of R_t on variables V_t is a solution of R_t on $\text{var}(R_t)$ and any assignment of the other variables.

Finally, $|Sol_{V_t}(R_t, \emptyset) \cap Sol_{V_t}(\bigvee_{j=1}^k \phi_j, \emptyset)| = \sum_{a \in R_t} S_{V_t}(\bigvee_{j=1}^k \phi_j, a)$ by Lemma 12. By Corollary 11, one can compute for each a the value $S_{V_t}(\bigvee_{j=1}^k \phi_j, a)$ in time $O(k)$ if the values of $S_{V_i}(\phi_i, a|_{V_i})$ are precomputed. But since the domain of a is $\text{var}(R_t)$, we have $a|_{V_i} = a|_{\text{var}(R_t) \cap V_i} = a|_{\text{var}(R_i)}$ by connectedness of the variables in the join tree \mathcal{T} . Thus $S_{V_i}(\phi_i, a|_{V_i}) = S_{V_i}(\phi_i, a|_{\text{var}(R_i)})$ which is precomputed by hypothesis.

Let $b' = b|_{\text{var}(R_t)}$. We compute $S_{V_t}(\phi_t, b')$ in the following way, similarly to before. We start with Lemma 8 to get

$$S_{V_t}(\phi_t, b') = S_{V_t}(R_t, b') + S_{V_t}(\bigvee_{j=1}^k \phi_j, b') - |Sol_{V_t}(R_t, b') \cap Sol_{V_t}(\bigvee_{j=1}^k \phi_j, b')|.$$

Again, by Corollary 11, one can compute $S_{V_t}(\bigvee_{j=1}^k \phi_j, b')$ in $O(k)$ if $S_{V_i}(\phi_i, b'|_{V_i})$ are known. But as the domain of b' is $\text{var}(R_t)$, $b'|_{V_i} = b'|_{\text{var}(R_i)}$ by connectedness of the variables in \mathcal{T} . So $S_{V_i}(\phi_i, b'|_{V_i})$ is precomputed since u is also an ancestor of t_i .

Moreover, $S_{V_t}(R_t, b') = \sum_{a \in R_t, b' \subseteq a} 2^{|V_t \setminus \text{var}(R_t)|}$ which can be computed in $O(|R_t|)$.

Finally, by Lemma 12, we have

$$|Sol_{V_t}(R_t, b') \cap Sol_{V_t}(\bigvee_{j=1}^k \phi_j, b')| = \sum_{a \in R_t, b' \subseteq a} S_{V_t}(\bigvee_{j=1}^k \phi_j, a).$$

And again, by Corollary 11, we can compute $S_{V_t}(\bigvee_{j=1}^k \phi_j, a)$ in time $O(k)$ if $S_{V_i}(\phi_i, a|_{V_i})$ is precomputed. For the same reasons as above, $a|_{V_i} = a|_{\text{var}(R_i)}$, thus these values were already computed by induction.

To conclude, we have seen that one can compute the $S_{V_t}(\phi_t, \emptyset)$ and $S_{V_t}(\phi_t, b|_{\text{var}(R_t)})$ for each $b \in R_u$ where u is an ancestor of t with $O(k \cdot r)$ arithmetic operations. Thus we can compute $S_{V_r}(\phi_r, \emptyset)$ in polynomial time. \square

4 Computing disjoint branches decompositions

In this section we will show how to compute disjoint branches decompositions of hypergraphs in polynomial time. We will first introduce *PQF*-trees, the datastructure that our algorithm relies on, then consider some structural properties of hypergraphs with disjoint branches decompositions and finally describe the algorithm itself, relying on objects we call *A*-separators.

4.1 PQF-trees

PQ-trees are a data structure introduced by Booth and Lueker [BL76] originally to check matrices for the so-called consecutive ones property. This problem can be reformulated as follows in our setting: Given a hypergraph $\mathcal{H} = (V, E)$, is there an ordering $\ell = e_1 \dots e_m$ of the edges such that if $v \in e_i \cap e_j$, then for all $i \leq k \leq j$, $v \in e_k$? We encode ordering

of edges by lists. We call such a list *consistent* for \mathcal{H} . Note that the notion of consistent lists matches exactly our notion of join paths.

A PQ -tree is a compact way of representing all the consistent lists for a hypergraph. We introduce a generalization of this data structure which we call PQF -trees.

Definition 14. Let $\mathcal{H} = (V, E)$ be a hypergraph. A PQF -tree for \mathcal{H} is defined to be an ordered tree with leaf set E such that

- the internal nodes are labeled with P , Q or F ,
- the P -nodes and F -nodes have at least two children, and
- the Q -nodes have at least 3 children.

A PQF -tree without F -nodes is called a PQ -tree.

PQF -trees will be used to encode sets of permutations of the edge set of a hypergraph that have certain properties. We write these permutations simply as (ordered) lists. To this end, we define some notation for lists and sets of lists. The concatenation of two ordered lists ℓ_1, ℓ_2 will be denoted by $\ell_1\ell_2$. If L_1, L_2 are two sets of lists, we denote by L_1L_2 the set $\{\ell_1\ell_2 \mid \ell_1 \in L_1, \ell_2 \in L_2\}$. With this notation we now define the notion of the frontiers of a PQF -tree which will be central to this section.

Definition 15. The frontiers $\mathcal{F}(T)$ of a PQF -tree T for $\mathcal{H} = (V, E)$ are a set of ordered list of the elements of E defined inductively by

- if T is a leaf e , then $\mathcal{F}(T) = \{e\}$,
- if T is rooted in t , having children t_1, \dots, t_k , then
 - if t is an F -node then $\mathcal{F}(T) = \mathcal{F}(T_1) \dots \mathcal{F}(T_k)$,
 - if t is a Q -node then $\mathcal{F}(T) = (\mathcal{F}(T_1) \dots \mathcal{F}(T_k)) \cup (\mathcal{F}(T_k) \dots \mathcal{F}(T_1))$,
 - if t is a P -node then $\mathcal{F}(T) = \bigcup_{\sigma \in \mathcal{S}_k} \mathcal{F}(T_{\sigma(1)}) \dots \mathcal{F}(T_{\sigma(k)})$ where \mathcal{S}_k is the set of permutations of $[k]$,

where T_i is the subtree of T rooted in t_i .

If for all $\ell \in \mathcal{F}(T)$, ℓ is a consistent list for \mathcal{H} , then we say that T is consistent for \mathcal{H} .

We recall the main theorem of [BL76], which allows to compute all possible join paths of a hypergraph in polynomial time.

Theorem 16 ([BL76]). Given a hypergraph (V, E) , one can compute in time $O(|E||V|)$ a PQ -tree T such that $\mathcal{F}(T)$ is exactly the set of consistent lists for \mathcal{H} .

In order to compute disjoint branches decompositions, we will need to compute join paths with additional restrictions. This is the reason for the introduction of F -nodes. It will be convenient to not have F -nodes that are children of other F -nodes and thus we introduce the following normal form for PQF -trees.

Definition 17. A PQF -tree T is said to be in normal form if there is no F -node in T having an F -node as a child.

Clearly, if an F -node t has a child u in T which is also an F -node, then we can remove u from T and connect its children to t without changing $\mathcal{F}(T)$. Thus we may always assume that all PQF -trees we encounter are in normal form.

We will in the remainder of this section use certain subtrees of PQF -trees which we call PQF -subtrees. These will be trees rooted in a vertex t of a PQF -tree, but they will not necessarily contain all descendants of t . Instead, we allow to “cut off” certain trees that are rooted by children of t . We now give a formal definition of PQF -subtrees. As usual, the *subtree rooted in t* is defined to be the tree induced by t and all its descendants.

Definition 18. Let T be a PQF -tree, and let t be a vertex of T . A subgraph S of T is said to be a PQF -subtree rooted in t if

- t is a leaf and S consists of the graph containing only t ,
- t is a P -node and S is the subtree rooted in t , or
- t is a Q -node or an F -node with children t_1, \dots, t_k and there exists i, j such that $1 \leq i < j \leq k$ and S is the graph containing t and T_i, \dots, T_k , the subtrees rooted in t_i, \dots, t_j .

We will now show that PQF -subtrees allow us to filter the frontier of a PQF -tree for certain lists that we will be interested in later. Remember that the *depth* of a node in a tree is its distance from the root.

Lemma 19. Let T be a consistent PQF -tree for (V, E) in normal form. Let $V' \subseteq V$ and $A = \{e \in E \mid V' \subseteq e\}$. Then there exists a PQF -subtree $T_{V'}$ of T such that the labels of the leaves of $T_{V'}$ are exactly A .

Proof. First assume that $V' = \{v\}$. Let t be the deepest node of T such that the set A is contained in the set of labels of the leaves of the subtree rooted in t .

If t is a leaf, then the PQF -subtree containing only t is the subtree we are looking for.

Otherwise, let t_1, \dots, t_k be the children of t . By maximality of the depth of t , we know that there are at least two children of t such that the subtrees rooted in them contain elements of A in their leaf labels. Let t_i be the leftmost such child and t_j the rightmost one. Note that $1 \leq i < j \leq k$. Let T_0 be the subtree of T rooted in t and T_1, \dots, T_k the subtrees rooted in t_1, \dots, t_k . Furthermore, we choose $\ell_s \in \mathcal{F}(T_s)$ for $s = 1, \dots, k$ arbitrarily. Note that the leaves of T_s are the edges in ℓ_s .

If t is a P -node then all the leaves of T_0 are in A . Indeed, suppose first that $i \neq 1$. As t is a P -node, there exists a list ℓ in $\mathcal{F}(T)$ having $\ell_i \ell_1 \ell_j$ as a sublist. By definition of i , the leaves in ℓ_1 do not contain v , but v is contained in some edges in ℓ_i and ℓ_j . Thus the connectedness condition for v is not respected in ℓ . This is a contradiction since T is consistent for \mathcal{H} . Analogously, we show $j = k$. Now, since t is a P -node, the lists $\ell_1 \ell_k$ and $\ell_k \ell_1$ are sublist of lists in $\mathcal{F}(T)$. Then because of the connectivity condition, the

first and the last edges of ℓ_1 and ℓ_k contain v . Thus all edges of ℓ_1 and ℓ_k are in A and consequently all edges of the subtree rooted in t .

If t is an F -node, we show that ℓ_i contains only edges in A . By connectedness of v , the last element of ℓ_i is in A . Moreover, as T is in normal form, t_i is not an F -node, thus the list ℓ'_i obtained by reversing ℓ_i is in $\mathcal{F}(T_i)$ and for the same reason, its last edges—that is the first of ℓ_i —is in A . It follows that all edges in ℓ_i are in A . Analogously, all edges in ℓ_j are in A , and by connectedness of v , all edges in the leafs of T_i, \dots, T_j are in A . Thus the leaves of the PQF -subtree rooted in t containing the subtrees T_i, \dots, T_j are exactly A , so this is the desired PQF -subtree.

If t is a Q -node, then $\ell_i \dots \ell_j$ and $\ell_j \dots \ell_i$ are sublists of a list in $\mathcal{F}(T)$. With the connectivity condition it follows that the first and the last edges of ℓ_i and ℓ_j are in A . Thus we find a PQF -subtree with the desired properties as before. This completes the case $V' = \{v\}$.

Now, if $V' = \{v_1, \dots, v_p\}$ we construct S iteratively. To this end, let $T_0 = T$ and for $i = 1, \dots, p$ we let T_i be the PQF -subtree of T_{i-1} whose leaves are exactly the edges containing v_{i+1} . The tree T_i can be found as argued above. Obviously, we have $T_{V'} = T_p$ which completes the proof. \square

During the construction of disjoint branches decompositions later, we will put restrictions on the position of some edges in join paths. To do so we will use the algorithm of the following proposition.

Proposition 20. *There is a polynomial time algorithm **Force** that, given a PQF -tree T and a PQF -subtree S of T , computes in polynomial time a PQF -tree $T' = \text{Force}(T, S)$ such that $\mathcal{F}(T') = \{\ell_1 \ell_2 \in \mathcal{F}(T) \mid \ell_2 \in \mathcal{F}(S)\}$. If this set is empty, the algorithm rejects.*

Proof. Let s be the root of the PQF -subtree S . We describe the algorithm by induction on the depth of s . Assume the depth of s is 0, that is if s is the root of T . If $S = T$, then we simply set $T' := T$. Otherwise, first observe that s is not a P -node since it would imply that $S = T$ by definition of PQF -trees. Let s_1, \dots, s_k be the children of s and $1 \leq i < j \leq k$ such that S is the (i, j) - PQF -subtree rooted in s .

If s is an F -node, then $\mathcal{F}(S) = \mathcal{F}(T_i) \dots \mathcal{F}(T_j)$ and $\mathcal{F}(T) = \mathcal{F}(T_1) \dots \mathcal{F}(T_k)$. Thus if $j < k$, then $\{\ell_1 \ell_2 \in \mathcal{F}(T) \mid \ell_2 \in \mathcal{F}(S)\}$ is empty since every $\ell \in \mathcal{F}(T)$ is of the form $\ell' \ell_k$ with $\ell_k \in \mathcal{F}(T_k)$ that is disjoint from the leaves in S . Thus the algorithm rejects in this case. However, if $j = k$ then $\mathcal{F}(S) = \mathcal{F}(T_i) \dots \mathcal{F}(T_k)$ and then for all $\ell \in \mathcal{F}(T)$, $\ell = \ell_1 \ell_2$ with $\ell_2 \in \mathcal{F}(S)$ (and $\ell_1 \in \mathcal{F}(T_1) \dots \mathcal{F}(T_{i-1})$). Consequently, T is the desired PQF -tree.

If s is a Q -node, $\mathcal{F}(S) = \mathcal{F}(T_i) \dots \mathcal{F}(T_j) \cup \mathcal{F}(T_j) \dots \mathcal{F}(T_i)$ and $\mathcal{F}(T) = \mathcal{F}(T_1) \dots \mathcal{F}(T_k) \cup \mathcal{F}(T_k) \dots \mathcal{F}(T_1)$. Thus, for the same reasons as in the last case, if $i \neq 1$ and $j \neq k$, there is no list in $\mathcal{F}(T)$ of the form $\ell_1 \ell_2$ with $\ell_2 \in \mathcal{F}(S)$ and the algorithm rejects.

Suppose that $j = k$. Let T' be the PQF -tree T where we replace s with an F -node. We have $\mathcal{F}(T') = \mathcal{F}(T_1) \dots \mathcal{F}(T_k)$. Just as before, we have $\mathcal{F}(T') \subseteq \{\ell_1 \ell_2 \in \mathcal{F}(T) \mid \ell_2 \in \mathcal{F}(S)\}$. For the other inclusion, as $S \neq T$, we also have $1 < i$. Let $\ell_1 \ell_2 \in \mathcal{F}(T)$ such that $\ell_2 \in \mathcal{F}(S)$. Thus $\ell_1 \in \mathcal{F}(T_1) \dots \mathcal{F}(T_{i-1})$ is not empty and thus $\ell_1 \ell_2 \in \mathcal{F}(T')$.

Now if $i = 1$, we reverse the children of s . This does not change $\mathcal{F}(T)$ and this case reduces to the previous one.

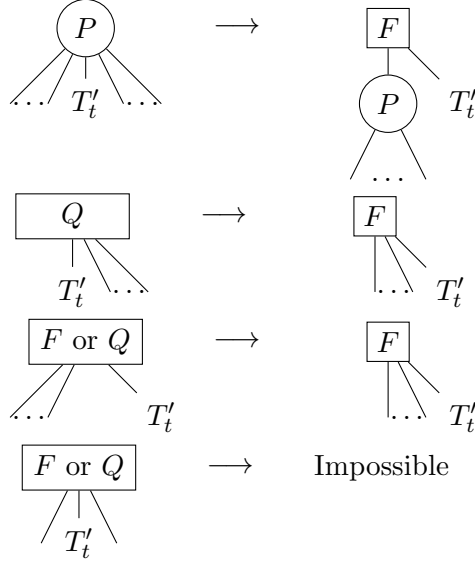


Figure 2: The transformations for the **Force** procedures

Now suppose that the depth of s is $d + 1$. Let r be the root of T and t the child of r such that the subtree T_t rooted in t contains S . In T_t , the depth of s is d . We recursively apply our algorithm on (T_t, S) to get a new PQF -tree T'_t such that $\mathcal{F}(T'_t) = \{\ell_1 \ell_2 \in \mathcal{F}(T_t) \mid \ell_2 \in \mathcal{F}(S)\}$. We claim that if we apply the transformation shown in the Figure 2 to the root r we get T' with the desired properties.

If the root r is a Q -node, for the same reasons as in the cases for depth 0, if t is not the rightmost or the leftmost child of r , then the transformation is impossible and the algorithm rejects. If not, we force the subtree T'_t on the right of the PQF -tree and the same reasoning as before will give the desired result.

If r is an F -node, the transformation is essentially the same as for Q -nodes except that we need t to be the rightmost child of r , since we cannot reverse the children here.

Let now r be a P -node. First observe that by induction hypothesis $\mathcal{F}(T'_t) = P\mathcal{F}(S)$ for some set P . Thus the set $\{\ell_1 \ell_2 \in \mathcal{F}(T) \mid \ell_2 \in \mathcal{F}(S)\}$ is obtained when we permute the children of P and bring T'_t on the right side. Thus we can apply any permutation on the $k - 1$ other children and let t on the right, which is exactly what the first transformation does.

Finally we see that we perform at most one change on each vertex lying between r and s , so the construction can easily be done in polynomial time. \square

Corollary 21. *Let $\mathcal{H} = (V, E)$ be a hypergraph and T a consistent PQF -tree for \mathcal{H} . Let $V' \subseteq V$ and $A = \{e \in E \mid e \cap V' \neq \emptyset\}$. Suppose that for all $e, f \in A$, $e \cap V' \subseteq f \cap V'$ or $f \cap V' \subseteq e \cap V'$. Then we can compute in polynomial time a PQF -tree T' such that $\mathcal{F}(T') = \{e_1 \dots e_m \in \mathcal{F}(T) \mid \forall i < j, e_i, e_j \in A \Rightarrow e_i \cap V' \subseteq e_j \cap V'\}$.*

Proof. We want to compute T' such that the frontiers of T' are the frontiers of T in

which the edges of A appear in increasing order with respect to inclusion relative to V' . The set $\{e \cap V' \mid e \in A\}$ is ordered by inclusion, thus it has a smallest element V_1 and a biggest element V_2 . Obviously, $V_1 \subseteq V_2 \subseteq V'$. Furthermore, $A = \{e \in E \mid V_1 \subseteq e\}$ and $A_2 = \{e \in A \mid V_2 \subseteq e\}$ is not empty.

First use Lemma 19 to find a PQF -subtree S of T whose leaves are exactly $\{e \mid V_1 \subseteq e\} = A$. Then use Lemma 19 again to find a PQF -subtree R of S whose leaves are exactly A_2 . Now use the procedure **Force**(S, R) to compute S' as in Proposition 20 and let T' be the tree where we replace S by S' in T . As finding the right subtrees in T can easily be done in polynomial time by finding a least common ancestor and **Force** is a polynomial time, it is clear that one can compute T' as well. We now show that T' has the desired properties.

To this end, let $\ell \in \mathcal{F}(T')$. By definition of T' , we have that ℓ is also in $\mathcal{F}(T)$. In addition, ℓ is of the form $\ell_1 \ell_A \ell_2$ with $\ell_A \in \mathcal{F}(S')$. By definition of **Force**, ℓ_A is of the form $\sigma_1 \sigma_2$ with $\sigma_2 \in \mathcal{F}(R)$, that is, consisting only of the edges in A_2 , which are maximal for the inclusion. Let $g \in A_2$. Let $e, f \in A$ with e appearing before f in ℓ_A . If $e \cap V'$ is not included in $f \cap V'$, then there exists a $v' \in V_2$ such that $v' \in e$, $v' \notin f$ and $v' \in g$ since $V_2 \subseteq g$. That would lead to an inconsistent list which is a contradiction.

Reciprocally, let $\ell \in \{e_1 \dots e_m \in \mathcal{F}(T) \mid \forall i < j, e_i, e_j \in A \Rightarrow e_i \cap V' \subseteq e_j \cap V'\}$. ℓ is of the form $\ell_1 \ell_A \ell_2$ with $\ell_A \in \mathcal{F}(S)$. As it is organized by inclusion relative to V' , the elements of A_2 should all lie at the end of ℓ_A . Thus $\ell_A \in \{\sigma_1 \sigma_2 \mid \sigma_2 \in \mathcal{F}(R)\} = \mathcal{F}(S')$ by Proposition 20. It follows that $\ell \in \mathcal{F}(T')$. \square

4.2 Db-rootable hypergraphs

In this section we will prove several structural properties of hypergraphs with disjoint branches decompositions which we will use in the algorithm in the next section.

Definition 22. Let $\mathcal{H} = (V, E)$ be a hypergraph. For $e \in E$, we say that \mathcal{H} is db-rootable in e if there exists a disjoint branches decomposition of \mathcal{H} rooted in e .

The algorithm for the construction of disjoint branches decompositions will delete edges of hypergraphs. To this end we introduce the following notation.

Definition 23. For a hypergraph $\mathcal{H} = (V, E)$ and an edge $e \in E$, we denote $\mathcal{H} \setminus e$ the hypergraph $(V_e, E \setminus \{e\})$ where $V_e := \bigcup_{e' \in E \setminus \{e\}} e'$. For a set $A = \{e_1, \dots, e_k\} \subseteq E$, we define $\mathcal{H} \setminus A$ to be the hypergraph $((\mathcal{H} \setminus e_1) \setminus \dots) \setminus e_k$.

We make the following observation which will simplify our arguments later.

Observation 24. Let $\mathcal{H} = (V, E)$ be a hypergraph and $e \in E$. Then \mathcal{H} is db-rootable in e if and only if for every connected component $C = (V_C, E_C)$ of $\mathcal{H} \setminus e$ the hypergraph $C' := (V_C \cup e, E_C \cup \{e\})$ is db-rootable in e .

The next lemma shows that deleting the root of a disjoint branches decomposition decomposes a hypergraph along the components of the resulting join forest.

Lemma 25. *Let \mathcal{H} be a hypergraph \mathcal{H} with a disjoint branches decomposition \mathcal{T} that is rooted in e . Let v_1, v_2 be two vertices that appear in different trees \mathcal{T}_1 and \mathcal{T}_2 of the forest $\mathcal{T} \setminus \{e\}$. Then v_1 and v_2 lie in different connected components of $\mathcal{H} \setminus e$.*

Proof. Suppose that there is a path e_1, \dots, e_k from v_1 to v_2 in $\mathcal{H} \setminus e$. We show by induction on i , that e_i is in \mathcal{T}_1 . It is true for $i = 1$ because $v_1 \in e_1$, the disjointness of \mathcal{T} prevents e_1 from being in \mathcal{T}_2 . Now, assume that $e_i \in \mathcal{T}_1$. As $e_i \cap e_{i+1} \neq \emptyset$, it means that e_i and e_{i+1} are in a common branch of \mathcal{T} . If e_i is an ancestor of e_{i+1} in \mathcal{T} , then by induction, e_{i+1} is in \mathcal{T}_1 . If e_{i+1} is an ancestor of e_i in \mathcal{T} , then either e_{i+1} is in \mathcal{T}_1 either $e_{i+1} = e$. However, $e_{i+1} \neq e$, because it is an edge of $\mathcal{H} \setminus e$, so $e_{i+1} \in \mathcal{T}_1$.

Thus $e_k \in \mathcal{T}_1$. However, it contradicts the disjointness of \mathcal{T} since $v_2 \in e_k$. Thus there is no path from v_1 to v_2 in $\mathcal{H} \setminus e$: they are in two different connected components. \square

Finally, we make one last observation on the roots of disjoint branches decompositions.

Lemma 26. *If \mathcal{H} is db-rootable in e and $\mathcal{H} \setminus e = (V', E')$ has one connected component then there exists $e' \in E'$ such that $e \cap V' \subseteq e'$.*

Proof. By Lemma 25, in a disjoint branches decomposition of \mathcal{H} rooted in e , the edge e has only one child e' . Thus, by connectedness, the vertices in e that are not in e' are only in e , thus they are not in V' by definition. So $e \cap V' \subseteq e'$. \square

4.3 Separators

By Observation 24 we may deal with the components of $\mathcal{H} \setminus e$ for a hypergraph \mathcal{H} and an edge e independently. Thus we will in this section always assume that $\mathcal{H} \setminus e$ just has a single component. We will consider restricted join paths that we call A -separators. In the following, all join paths will be denoted as ordered lists of edges, which corresponds to the notation in Section 4.1.

Definition 27. *Let $\mathcal{H} = (V, E)$ be a hypergraph and let $\mathcal{P} = a_1 \dots a_m$ be a join path of $A \subseteq E$. We call \mathcal{P} an A -separator of \mathcal{H} if for all connected components $C = (V_C, E_C)$ of $\mathcal{H} \setminus A$ we have that if $a_j \cap V_C \neq \emptyset$, then for all $i \leq j$, $a_i \cap V_C \subseteq a_j \cap V_C$.*

Theorem 28. *There is a polynomial time algorithm `ComputeSeparator`(\mathcal{H}, A) that, given a hypergraph $\mathcal{H} = (V, E)$ and a set $A \subseteq E$, computes an A -separator of \mathcal{H} if it exists and rejects otherwise.*

Proof. We will iterate the algorithm described in Corollary 21. We first compute a PQ -tree T_0 such that $\mathcal{F}(T_0)$ is the set of all join paths for A using Theorem 16. Then, for each component $C = (V_C, E_C)$ of $\mathcal{H} \setminus A$, we iteratively do the following: If there are edges a_i and a_j such that $a_i \cap V_C \not\subseteq a_j \cap V_C$ and $a_j \cap V_C \not\subseteq a_i \cap V_C$, then \mathcal{H} cannot have an A -separator and we reject. Otherwise, we can use the algorithm of Corollary 21 on the PQF -tree we have computed so far to construct a PQF -tree whose frontiers respect the order condition for the edges imposed by C or rejects.

An easy induction shows that if this algorithm does not reject at any point, then the computed PQF -tree T has as frontiers all join paths that satisfy the order conditions

of Definition 27. Thus we can choose one of these join paths arbitrarily as the desired A -separator. \square

We will need A -separators with additional conditions.

Definition 29. Let $\mathcal{H} = (V, E)$ be a hypergraph and let $A \subseteq E$. We call an A -separator $\mathcal{P} = a_1 \dots a_m$ of \mathcal{H} a strong A -separator if for all connected components $C = (V_C, E_C)$ of $\mathcal{H} \setminus A$, $C' = (V_C, E_C \cup \{a_{l_C}\})$ is db-rootable in a_{l_C} where $l_C = \max\{i \mid a_i \cap V_C \neq \emptyset\}$.

As we will see, the existence of a strong A -separator for $\mathcal{H} \setminus e$ and a certain set A is equivalent to the property of being db-rootable in e . Thus we are interested in computing strong A -separators instead of arbitrary A -separators. Fortunately, it turns out that if there exists a strong A -separator, then all A -separators are strong and thus the algorithm of Theorem 28 suffices.

Proposition 30. Let $\mathcal{H} = (V, E)$ be a hypergraph and $A \subseteq E$. If there exists a strong A -separator of \mathcal{H} then all A -separators of \mathcal{H} are strong.

Proof. Let \mathcal{P} be an A -separator and C a connected component of $\mathcal{H} \setminus A$. Let e_C be the edge of the existing strong separator in which $C \cup \{e_C\}$ is db-rootable. Let \mathcal{T}_C be the corresponding disjoint branches decomposition rooted in e_C and let f_C be the last edge on \mathcal{P} such that $f_C \cap V_C \neq \emptyset$. As $e_C \cap V_C \neq \emptyset$, e_C comes before f_C on \mathcal{P} . As \mathcal{P} is an A -separator, $e_C \cap V_C \subseteq f_C \cap V_C$. As only vertices of V_C appear in the children of e_C in \mathcal{T}_C , we can replace the root e_C of \mathcal{T}_C without breaking the connectedness condition of the vertices of V_C or the disjointness of the branches (we do not change the branches of \mathcal{T}_C). Doing this for all components of $\mathcal{H} \setminus C$ shows that \mathcal{P} is strong. \square

We will now show the main theorem that directly yields the algorithm for the computation of disjoint branches decomposition by reducing the construction of a disjoint branches decomposition to the construction of strong separators.

Theorem 31. Let $\mathcal{H} = (V, E)$ be a hypergraph, $e \in E$, and $\mathcal{H}' = (V', E') = \mathcal{H} \setminus e$. Let furthermore $A_e := \{e' \in E' \mid e \cap V' \subseteq e'\}$. Assume that \mathcal{H}' has only one connected component. Then \mathcal{H} is db-rootable in e if and only if there exists a strong A_e -separator of \mathcal{H}' .

Proof. Suppose \mathcal{H} is db-rootable in e . Let \mathcal{T} be a disjoint branches decomposition of \mathcal{H} rooted in e . By Lemma 26, A_e is not empty. Let $v \in e \cap V'$. We know that v is contained in all edges $e' \in A_e$. Thus, by disjointness, the edges in A_e are on the same branch of \mathcal{T} . Moreover, we claim that A_e is connected in \mathcal{T} . To see this, suppose that $b \in E$ is between $a, c \in A_e$ on this branch. Then by connectedness, $e \cap V' \subseteq b$, so $b \in A_e$. Consequently, A_e is connected and thus forms a path. Let $\mathcal{P} = a_1 \dots a_k$ be this path in \mathcal{T} in the direction from the root to the leaves of the tree. We claim that \mathcal{P} is a strong A_e -separator.

To this end, let $C = (V_C, E_C)$ be a connected component of $\mathcal{H}' \setminus A_e$. We consider the forest obtained by removing A_e in \mathcal{T} . By Lemma 25, vertices in different trees of this forest are in different connected component of $\mathcal{H}' \setminus A_e$ as well. Thus there is a tree \mathcal{T}_C

that contains all the edges in E_C . Let a_{l_C} be the edge of A_e to which the root of \mathcal{T}_C is connected. If $j > l_C$, then we claim that $a_j \cap V_C = \emptyset$. Assume this were not the case, then \mathcal{T}_C and the subtree of a_{l_C} containing a_j were not disjoint which is a contradiction to \mathcal{T} being a disjoint branches decomposition. Thus $a_j \cap V_C = \emptyset$. Now consider $i < j \leq l_C$ and let $v \in a_i \cap V_C$. We have $v \in a_{l_C} \cap V_C$ by the connectedness condition since v appears in \mathcal{T}_C and, again by connectedness the connectedness condition, $v \in a_j \cap V_C$. Thus, $a_i \cap V_C \subseteq a_j \cap V_C$. It follows that \mathcal{P} is an A_e -separator. As $C \cup \{a_{l_C}\}$ is db-rootable in a_{l_C} using the tree \mathcal{T}_C , we have that \mathcal{P} is a strong A_e -separator.

Assume now that there is a strong A_e -separator $\mathcal{P} = a_1 \dots a_k$. For a connected component C of $\mathcal{H}' \setminus A_e$, let $l_C := \max\{i \mid a_i \cap V_C \neq \emptyset\}$. By definition, there exists a disjoint branches decomposition \mathcal{T}_C for $C \cup \{a_{l_C}\}$ rooted in a_{l_C} . We construct a disjoint branches decomposition for \mathcal{H} as follows:

- We root the path $ea_1 \dots a_k$ in e .
- For each connected component C of $\mathcal{H}' \setminus A_e$, we connect the root of \mathcal{T}_C to a_{l_C} .

We claim that the resulting tree \mathcal{T} is a disjoint branches decomposition. We first show that the branches of \mathcal{T} are disjoint. Indeed, if a, b are edges in two different branches then two cases can occur: Either a and b are in two different connected component of $\mathcal{H}' \setminus A_e$. But then they are disjoint, because they lie in different components of \mathcal{H} by construction. Otherwise, let a be in a connected component C of $\mathcal{H}' \setminus A_e$ and let b be in A_e . But as b is on a different branch as a , it follows that b comes after a_{l_C} on \mathcal{P} . Thus $b \cap V_C = \emptyset$ and it follows that $a \cap b = \emptyset$ since $a \subseteq V_C$. Thus the branches of \mathcal{T} are disjoint.

Now we show that \mathcal{T} is a join tree, i.e., it satisfies the connectedness condition for all vertices. For a connected component C of $\mathcal{H}' \setminus A_e$, for a vertex $v \in V_C$, its connectedness is ensured along \mathcal{P} since \mathcal{P} is a join path and in \mathcal{T}_C since \mathcal{T}_C is a join tree. Furthermore, by construction $v \in e_{l_C}$ so the edges containing v are connected in \mathcal{T} . For a vertex v , which does not appear in any V_C , that is, which only appears in A_e , its connectedness is ensured by the fact that \mathcal{P} is a join path for A_e . \square

We now turn the proof of Theorem 31 into the algorithm for the computation of disjoint branches decompositions.

Corollary 32. *There is a polynomial time algorithm $\text{ComputeDB}(\mathcal{H}, e)$ that, given a hypergraph $\mathcal{H} = (V, E)$ and an edge $e \in E$, returns a disjoint branches decomposition of \mathcal{H} rooted in e if it exists and rejects otherwise.*

Proof. We give a pseudo-code of ComputeDB in Algorithm 1. Following Observation 24, we deal with the connected component of $\mathcal{H} \setminus e$ independently. For each connected component of \mathcal{H} , we compute A_e . We reject if it is empty since it means by Lemma 26 that \mathcal{H} is not db-rootable in e . Otherwise we compute an A_e -separator $\mathcal{P} = a_1 \dots a_k$ of \mathcal{H} in polynomial time with the procedure ComputeSeparator of Theorem 28 and recursively check its strongness by trying to compute a disjoint branches decomposition \mathcal{T}_C of $C \cup \{a_{l_C}\}$ rooted in a_{l_C} where $l_C = \max\{i \mid a_i \cap V_C \neq \emptyset\}$. Following the proof of

Theorem 31, we construct a disjoint branches decomposition of \mathcal{H} rooted in e by rooting the path $ea_1 \dots a_k$ in e and connecting the root of \mathcal{T}_C to a_{l_C} .

Theorem 31 ensures both that if the algorithm does not reject then the computed decomposition is a disjoint branches decomposition of \mathcal{H} rooted in e and that if such a decomposition exists then the algorithm won't reject.

Algorithm 2 The algorithm `ComputeDB` of Corollary 32.

```

ComputeDB( $\mathcal{H} = (V, E)$ ,  $e$ ) =
if  $|E| = 1$  then return the tree with the only vertex  $e$ 
else
  for each connected component  $\mathcal{H}_i = (V_i, E_i)$  of  $\mathcal{H} \setminus e$  do
     $A_e \leftarrow \{e' \in E_i \mid e \cap V_i \subseteq e'\}$ 
    if  $A_e = \emptyset$  then
      Fail.
     $\mathcal{P} \leftarrow \text{ComputeSeparator}(\mathcal{H}_i, A_e)$ 
     $\mathcal{T}_i \leftarrow \mathcal{P}$ 
    for each connected component  $C = (V_C, E_C)$  of  $\mathcal{H}_i \setminus A_e$  do
       $l_C \leftarrow \max\{j \mid a_j \cap V_C \neq \emptyset\}$  (where  $\mathcal{P} = a_1 \dots a_k$ )
       $C' \leftarrow (V_C \cup a_{l_C}, E_C \cup \{a_{l_C}\})$ 
       $\mathcal{T}_C \leftarrow \text{ComputeDB}(C', a_{l_C})$ 
      connect  $\mathcal{T}_C$  to  $\mathcal{T}_i$  in  $a_{l_C}$ 
  return the tree rooted in  $e$  having  $\mathcal{T}_1, \dots, \mathcal{T}_p$  as children

```

The algorithm runs in polynomial time since there is at most $|E|$ recursive calls of `ComputeDB` and that each call is in polynomial time. \square

5 Some negative results on generalizations

In this section we will discuss several approaches to generalizing the counting algorithm for hypergraphs with disjoint branches to more general classes of hypergraphs. Unfortunately, all these results will be negative as we will show hardness results for all extensions we consider. We still feel these results are worthwhile because they might help in guiding future research to classes of hypergraphs that are better suited for $\#SAT$.

Our main technical tool will be a helpful result by Samer and Szeider [SS10]. For a hypergraph $\mathcal{H} = (V, E)$, let \mathcal{H}^* be the hypergraph $\mathcal{H}^* := (V \cup \{x\}, E \cup \{V \cup \{x\}\})$ where x is a new vertex. We use the following result of Samer and Szeider.

Lemma 33 ([SS10]). *Let F be a CNF-formula with hypergraph \mathcal{H} . Then we can in polynomial time construct a CNF-formula F' with hypergraph \mathcal{H}^* such that*

- *F has a satisfying assignment if and only if F' has one, and*
- *the number of satisfying assignments of F can be computed from those of F' in polynomial time.*

Note that Lemma 33 is not proved explicitly in [SS10] but follows from the proof of Proposition 1 in that paper.

5.1 Undirected paths

In this section we consider a generalization of disjoint branches in the following way: We call a hypergraph \mathcal{H} *undirected path acyclic* if there is a join tree (\mathcal{T}, λ) of \mathcal{H} such that for every $v \in V$ the edge set $\{e \in E \mid v \in e\}$ forms an undirected path in \mathcal{T} . Undirected path acyclicity is a seemingly natural generalization of disjoint branches acyclicity by allowing undirected paths instead of directed paths. Unfortunately, as we will see this generalization makes our counting problem hard and in fact even SAT hard. To show this we will use the following notions: We call a CNF-formula *read-twice* if every variable appears at most twice in it.

Lemma 34 ([IS90]). *SAT for read-twice-formulas is NP-hard.*

We show that undirected path acyclicity does not not allow tractable SAT and thus it is not a good generalization in our setting.

Theorem 35. *SAT on undirected path acyclic CNF-formulas is NP-complete.*

Proof. Let F be a read-twice-formula. We construct F' as in Lemma 33 and claim that F' is undirected path acyclic. The corresponding join tree has $V \cup \{x\}$ as its root and all other edges are leaves of the join tree that are connected to the root. Then every variable appears in at most 2 leaves because F is read-twice. Since all variables also appear in the root, every variable induces a path in this join tree. So F' is undirected path acyclic.

The claim is now easy to see: SAT for F and F' is equivalent, but with Lemma 34 SAT for F is hard. This gives the desired reduction for NP-hardness. \square

5.2 Allowing limited intersections and appearance in several branches

A natural way of generalizing disjoint branches decompositions is allowing limited intersections between branches. We will show that this approach leads to hard counting problems even if we only allow the intersection to contain one variable.

Lemma 36. *#SAT is #P-hard for CNF-formulas that have a join tree in which the branches may have a pairwise intersection containing one variable.*

Proof. We reduce from #VertexCover which can alternatively be interpreted as #2-SAT on monotone formulas and is well known to be #P-hard. So let F be a monotone 2-CNF formula. We construct F' as in Lemma 33. Let (\mathcal{T}, λ) be a join tree of F' in which the edge $\text{var}(F')$ is the root and all other edges are leaves. Since F is monotone and we may assume that no clause appears twice in it, the leaves intersect in at most one variable which completes the proof. \square

6 Conclusion

We have presented a new structural class of tractable #SAT instances, those whose hypergraphs admit a disjoint branches decomposition. To this end, we also invested a considerable amount of work into an algorithm that computes the decompositions.

Several questions remain, the most obvious open problem certainly being the complexity of #SAT on β -acyclic hypergraphs. Can one show a #P-completeness result or a polynomial time algorithm for this case?

Another aim for future work is trying to turn the disjoint branches property into a hypergraph width measure such that #SAT—or even SAT—for the hypergraphs for which this width measure is bounded is tractable? Can we construct this measure to even allow fixed-parameter tractability? Note that it is known that the parameterization by incidence cliquewidth both does not allow fixed-parameter tractability [OPS13].

More generally, we feel that it is very desirable to understand the tractability frontier for SAT and #SAT with respect to structural restrictions better overall. Is there a width measure that generalizes both hypergraphs with disjoint branches and incidence cliquewidth that leads to tractable #SAT? A natural candidate would be β -hypertree width (see Figure 1). Are there other classes of hypergraphs incomparable to those studied so far that give large structural classes of tractable #SAT-instances? Note that there is a similar line in the area of constraint satisfaction (see e.g. [Mik08] for an overview) that has been very successful but unfortunately does not apply directly.

References

- [BB12] J. Brault-Baron. A Negative Conjunctive Query is Easy if and only if it is Beta-Acyclic. In *Computer Science Logic, 26th International Workshop/21st Annual Conference of the EACSL*, pages 137–151, 2012.
- [BL76] K.S. Booth and G.S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, 1976.
- [BL03] A Brandstädt and Vadim V Lozin. On the linear structure and clique-width of bipartite permutation graphs. *Ars Combinatoria*, 67(1):273–281, 2003.
- [CGH09] D.A. Cohen, M.J. Green, and C. Houghton. Constraint representations and structural tractability. In *Proceedings of the 15th international conference on Principles and practice of constraint programming*, CP’09, pages 289–303, 2009.
- [Dur12] David Duris. Some characterizations of γ and β -acyclicity of hypergraphs. *Inf. Process. Lett.*, 112(16):617–620, 2012.
- [Fag83] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM*, 30(3):514–550, 1983.

- [FG06] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer-Verlag New York Inc, 2006.
- [FMR08] E. Fischer, J.A. Makowsky, and E.V. Ravve. Counting truth assignments of formulas of bounded tree-width or clique-width. *Discrete Applied Mathematics*, 156(4):511–529, 2008.
- [GP04] G. Gottlob and R. Pichler. Hypergraphs in Model Checking: Acyclicity and Hypertree-Width versus Clique-Width. *SIAM Journal on Computing*, 33(2), 2004.
- [IS90] H.B. Hunt III and R.E. Stearns. The Complexity of Very Simple Boolean Formulas with Applications. *SIAM Journal on Computing*, 19(1):44–70, 1990.
- [Mik08] Z. Miklós. *Understanding Tractable Decompositions for Constraint Satisfaction*. PhD thesis, University of Oxford, 2008.
- [OPS13] S. Ordyniak, D. Paulusma, and S. Szeider. Satisfiability of acyclic and almost acyclic CNF formulas. *Theoretical Computer Science*, 481:85–99, 2013.
- [PSS13] D. Paulusma, F. Slivovsky, and S. Szeider. Model Counting for CNF Formulas of Bounded Modular Treewidth. In *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013*, pages 55–66, 2013.
- [Rot96] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1–2):273 – 302, 1996.
- [Sli14] F. Slivovsky. personal communication, 2014.
- [SS10] M. Samer and S. Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50–64, 2010.
- [SS14] F. Slivovsky and S. Szeider. Model Counting for Formulas of Bounded Clique-Width. to appear, 2014.